Dynamic Scalable Distributed Face Recognition System Security Framework

Konrad Rzeszutek

April 29, 2002

Abstract

Many of our daily interactions depend on being able to recognize one's face. Modeling human recognition can be used in many fields: surveillance systems, security systems, autonomous navigation of vehicles and many more.

Current face recognition technologies are extremely computationally intensive. To accommodate the load in near-real time, a multi component scalable distributed framework is presented. This framework allows separating the functionality of recognition, notification, and replay of camera-feed in different independent components. Furthermore the framework was designed with RAS (reliability, availability and serviceability), adaptability to network changes and easy path of adding (or replacing) of components in mind.

This paper describes the framework, detailed explanation of one of the face recognition technologies used in the work, motion detection technology used in capture system and future work.

Chapter 1

Dynamic Scalable Distributed Face Recognition Security System Framework

The current advances in computer technology has resulted in manufacturing inexpensive, high-capacity, high-computationally-feasible systems. A cluster of these systems is capable of outpacing some of the high-performance supercomputers at a fraction of cost. Face recognition, belonging to the family of computationally intensive applications, requires high-performance systems to perform its calculations. To solve the calculations in near real-time would require expensive high-performance supercomputers or clustering inexpensive systems.

Therefore a scalable distributed multi-component framework system called Apollo is proposed. This system would augment the computational intensity requirments of face recognition matching on a large scale. Furthermore Apollo was designed with reliability, availability, and serviceability (RAS) in mind. The reliability lies in its dynamicity and adaptability traits in changing network environment. The off-site storage and redundancy fulfill the availability profile and the serviceability lies in its scalable feature and option in replacing (or upgrading) components.

1.1 Definition of terms

- *Pool* - The collection of machines in one of the components.

- *Systems* - A collection of computer systems.
- *Services* - A set of computer services (such as web server, lookup server).
- *Clients* - Clients using the services (web browser).
- *RMI API* - Remote Method Invocation library (Java's portable answer to RPC - Remote Procedure Call).
- *Jini* - From Jini Network Technology Datasheet [11]: Network API enabling the spontaneous assembly and interaction of services on a network.
- *JMF* - From JMF FAQ [9]: "Java Media Framework API specifies a simple, unified architecture to synchronize and control audio, video and other time-based data within Java applications and applets."
- *RTP* - From RTP specification (RFC 1889) [4]: "RTP provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio, video or simulated data, over multicast or unicast network services."
- *Off-line storage* - Storing of camera-feed on a redudant system for latter replay if neccesary.

1.2 Components of Apollo

There are five components, which are:

- *Hermes* - the load balancer acting as *police officer* directing traffic to the appropriate components capable of handling the required requests and data.
- *Ares* - the thin client sending its camera feed to off-site storage and face recognition (Demeter and Nemesis respectively).
- *Demeter* - the off-site storage component. Stores the camera feed for future replay.

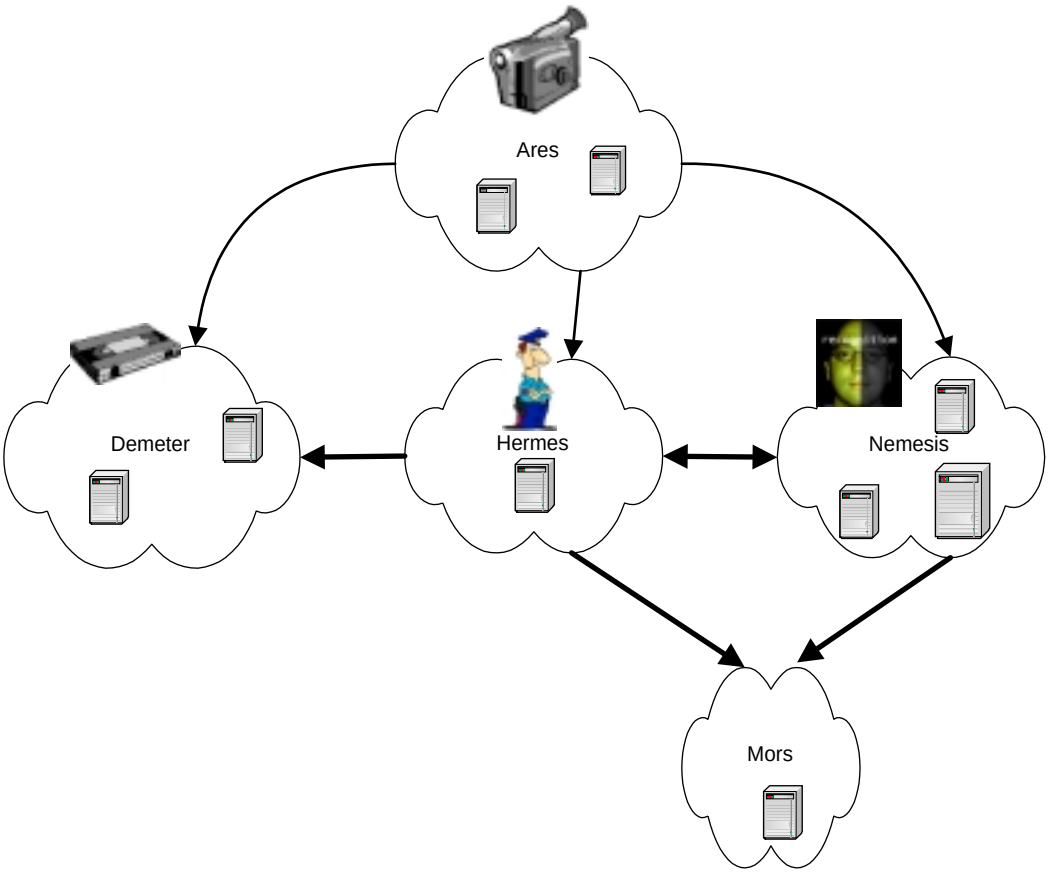

Figure 1.1: Components of Apollo

- *Nemesis* - the number crunching component - processes the images in an attempt to recognize a face.
- *Mors* - the logging / notifying component saving the trigger events (a face is recognized).

1.3 Hermes

Hermes - the messenger of Gods, though in this scenario plays a role of the "police officer" directing traffic (see Fig: 1.1). Hermes acts as dynamic centralized information portal. Since the system is distributed many systems can exist in a Hermes pool and provide information to its clients. The information is queried directly from the other three components: Demeter, Nemesis and Mors at every predefined amount of time. This allows for near real-time acquisition of load information, number of clients using the component, the maximum number of connections allowed and of course the address of the component.

To have such a highly dynamic system, capable of detecting changes in such dynamic computing environment, Hermes must be quite adaptive. Furthermore, Hermes by itself must be scalable too (its part of a distributed system) - therefore there can be many systems in a Hermes component pool. Each of them has the same information about the three other components, though each retrieves the information by itself. To find other components and to be found, it uses Jini technology, which is described in more details in Chapter 4.

The main purpose of Hermes is to provide information for two components: Ares and Nemesis. Ares requires information about Nemesis and Demeter, while Nemesis requires information about Mors (see Fig: 1.5). Hermes provides this information - giving the client the least loaded requested component, therefore eliminating bottlenecks and keeping a fairly level load across the systems. When the load on the specific pool of components if fairly high, the solution is to add a system of that component, and the load balance will direct new traffic to that system.

In summary Hermes' solely task is to provide the addresses of the desired components with the least load.

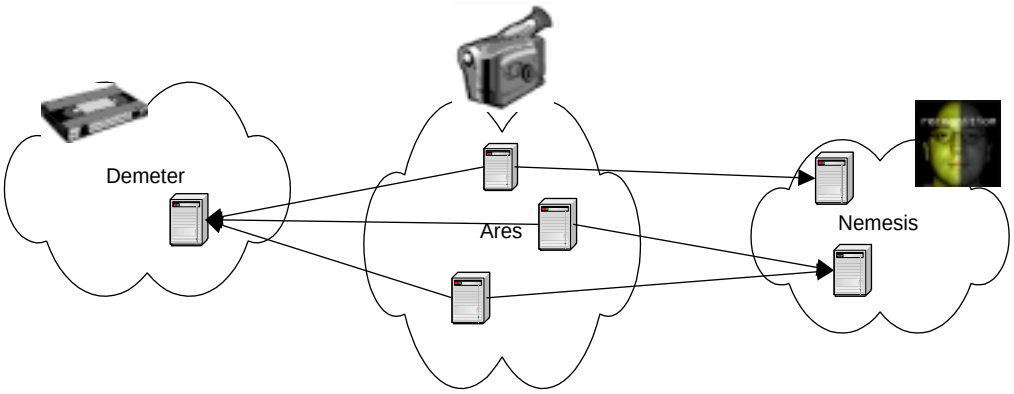

Figure 1.2: Ares interacting with Demeter and Nemesis

1.4 Ares

Ares - the God of War - is responsible for capturing the data feed from the camera and sending it to an off-site storage and face recognition component (Demeter and Nemesis respectively) (see Fig: 1.2). Since the system is dynamic, it has to be capable of finding the required components dynamically. To do so, it locates Hermes - the load balancer - and queries for the desired component. Upon receiving the addresses, it registers itself with these two components and starts sending its camera feed (see Fig: 1.3). On a side note it's worth noting that the camera feed is first processed through motion detection plug-in to save on bandwidth and time-stamped with data and location. Also the camera feed is stored locally, for redundancy reasons.

And that is the extent of the Ares functionality. It's rather a dull component, more like an eye in a human - the eye by itself cannot do any processing, but the brain does it.

The transportation mechanism used to transfer the camera feed is RTP multicast. More information about RTP is found in the "RFC 1889" and "JMF FAQ" [4, 9] and examples of how to work with in Java and JMF in "Java(TM) Media Framework API Guide" [10].

It is also modular - if this piece fails, there are many other ones that can take the job over, thought they might not be in the same physical place.

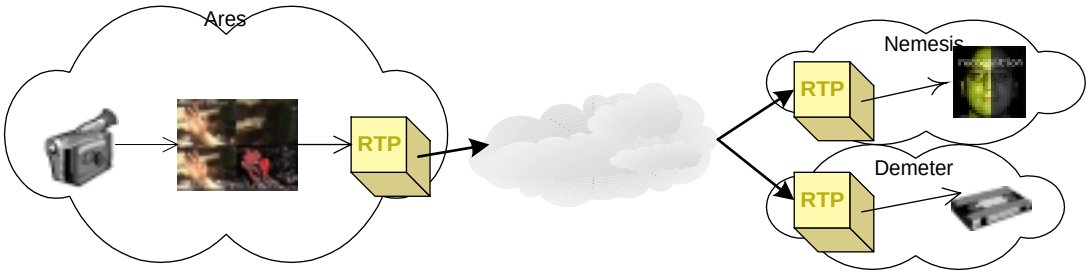

Figure 1.3: Ares frames passing through the motion detection engine, handled off by RTP to be distributed to Nemesis (for face recognition) and Demeter (for storage).

1.5 Demeter

Demeter - the goddess of harvest - is responsible for storing the camera feed from Ares (see Fig: 1.4). It's a repository serving to collect in one pool the camera feed from various Ares components. It's intended to allow the generation of time-lapse movies from each individual Ares. This allows for precise replaying the camera feed when a face was recognized from a centralized location.

Since this component requires vast storage capacity, a pool of systems is required. Therefore the Demeter's pool is scalable - as the need increases, more systems are added into the pool. Also machines can be removed and added dynamically (for hardware upgrades for example) from the pool.

In summary, Demeter is a modular piece of the distributed system. This component stores the camera feed (which nota bene is also being done on Ares) - and further provides another functionality: a centralized location to replay the camera feed whenever required. The camera-feed is received using RTP.

1.6 Nemesis

Nemesis - the Greek Goddess of vengeance who punished those who had broken the moral code. This component contains the face-recognition engine. It receives the camera feed from Ares and compares the image to the ones in its database (see Fig: 1.5). Since the face recognition task on a large scale

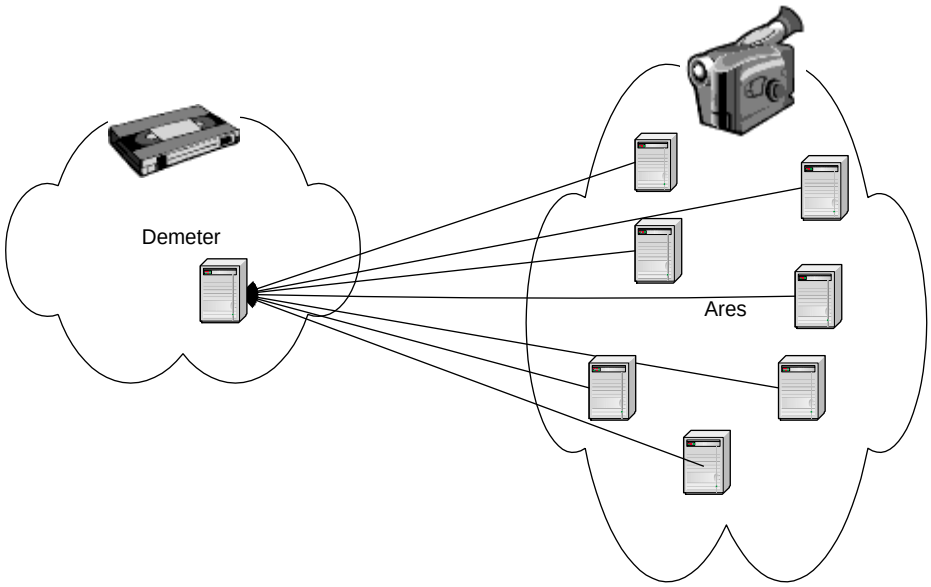

Figure 1.4: Demeter interaction with multiple systems in Ares

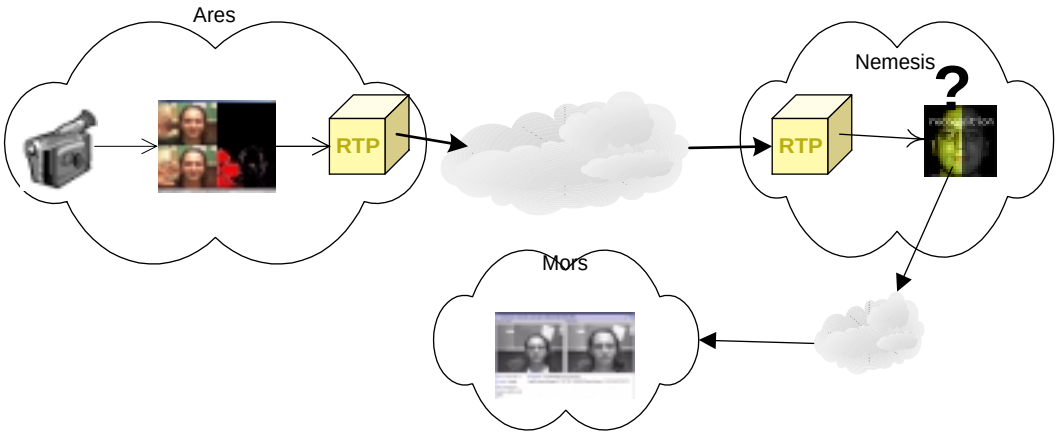

Figure 1.5: Nemesis interaction with Ares and Mors

Figure 1.6: Event from Nemesis

of images is quite computationally intensive, this component requires a hefty pool of high-performance machines (preferable). The images not processed immediately are buffered. This allows for the later matching of faces. When a match occurs, an *event* is sent to Mors along with the image from the camera feed, the matching image, date, and location of match (see Fig: 1.6). This event is also saved locally for redundancy purpose.

The under-laying face recognition technology is a quite computationally intensive task. To accommodate high inflow of camera feed among these systems the high inflow of data needs to be distributed across the pool. It is also worth noting it is possible to have a number of face recognition engines in the Nemesis component. This would allow for finding a match that intersects the results of the face recognition engines.

The mechanism for finding Mors component is the same as in all other components - by contacting Hermes, the load balancer and finding the least loaded Mors server. The transport mechanism to receive the camera feed is the same as in Demeter - RTP.

In summary, Nemesis is the most important component of this framework - it does the face recognition and notifies other components about possibly matches.

1.7 Mors

Mors - (aka Thanatos) - personification of death - is the component that receives face recognition events. This component is responsible for showing the operator (the user) that a face-recognition match between an image in

the Nemesis face database and the camera feed from Ares occurred. The location of the camera, along with date, the camera image and the matched image is submitted to Mors.

In summary, Mors provides a centralized pool where events are recorded. Having a single point (or many single points) where events are stored facilities the instant comparison of the matches with the camera's feed. This allows for humans to verify the result and act accordingly if there is a need.

1.8 How they work together

Each system is autonomous - in a component pool each machine is completely independent of each other. However, not each pool is independent of each other. Nemesis communicates with Hermes and Mors. Ares communicates with Hermes, Nemesis and Demeter.

Hermes, being the "police officer" is the most critical component. Without Hermes presence, Ares would be unable to find its required components and newly started services would be unable to find their required services. Thought components that already have found their services and are communicating are not affected (unless one of the components fails and its necessary to find an replacement). It's worth noting that having multiple systems in Hermes pool is not a problem. Each of these Hermes systems will have the same information about the different pools, albeit due to network latency, information might be temporarily out of synchronization.

1.9 Primary operation of Apollo

There are many operations in this system. Each component by itself performs internally many functions. Detailed explanation of face recognition will be explained in Chapter 3, motion detection in Chapter 2 and Chapter 4 will explain the dynamic aspects of the components.

This section rather focuses on the primary operations of a face recognition system (see Fig: 1.7, which are:

- How a frame captured by the camera traverses through the system.
- What happens to it when it matches to the internal database of images?
- How is the camera feed saved on an off-line storage?

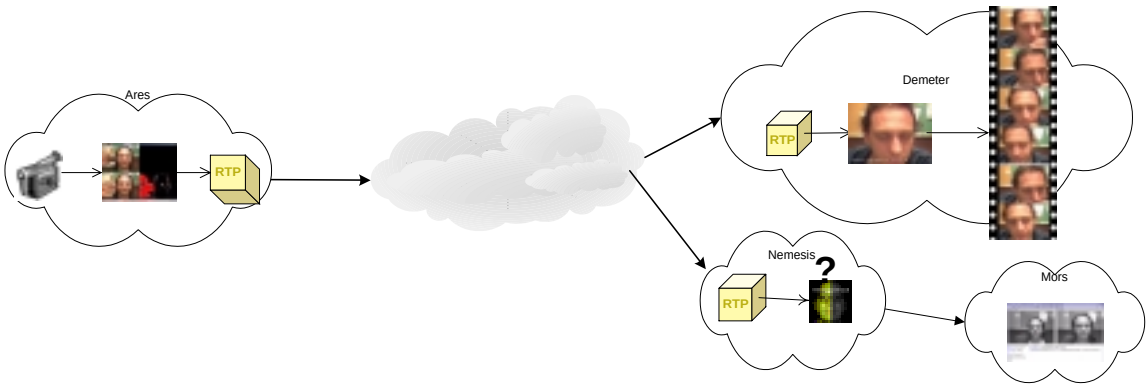

Figure 1.7: Primary operation of Apollo

The camera feed is handled by one component - Ares, Ares receives the camera feed and based on motion detection engine either passes the frames on or drops them. If the frame passed the check, the frame is sent to the next two components - Demeter and Nemesis. When Demeter and Nemesis received it, they saved it locally. It's assumed that Ares has already obtained the address of Nemesis and Demeter from Hermes.

Demeter stores the frames until a predefined amount of time has elapsed (usually twenty four hours) and at which point it generates a movie from the stored frames and deletes them.

Nemesis buffers the received frames and at some predefined period of time tries to match the frames to its internal database of pictures. If there is no match the frame is expunged. Otherwise an event (see Fig: 1.6) is generated which is saved locally and propagated to Mors. Mors upon receiving the message saves it locally and displays the event to the operator.

1.10 Other work related in this field

The author has not found any distributed computation solution to the face recognition technology. Only autonomous systems with the training images and the camera feed locally have implemented.

This work presents a different paradigm of matching faces in a near real-time time scale. It also provides further expandability - such as replacing the face recognition technology or augmenting it.

1.11 Motivation

The idea arose after the September 11th attack on United States by Al Qaeda terrorist network. The author found that the current technologies used in screening passengers are completely useless and only offer to calm the general population in believing that random searches of young good-looking women will reveal the terrorist.

Therefore the idea a non-intrusive passive surveillance system capable of processing the information against a large database in a distributed scalable system was envisioned. The current face recognition technologies are computationally intensive and if the system were to use a number of face recognition technologies the tool would be quite overpowering for autonomous system. While in a distributed scalable system this could work. This thesis shows how such could be designed, its inner details and a working example.

1.12 Summary

With this dynamic distributed modular component system, where each component has a specific predefined function, upgrading and replacing components is a simple task. Therefore upgrading the face recognition component for a better one is quite simple. Also the components can be removed - the off-site storage component and event logger do not have necessarily have to be present. They offer only offer redundancy and a centralized pool to receive communication. Furthermore the distributed approach allows for computationally demanding face recognition computations. With thin clients serving as "eyes", distributed systems processing the image and others storing the image it resembles the way a human brain processes images and remembers some of them.

Chapter 2

Motion Detection

The motion detection engine is used in the thin client (Ares). The engine processes each frame from the camera feed and passes it to the next stage (determined by Ares) if it has *enough* motion in it. The enough deferential is a threshold function. But before the threshold function can be applied, the incoming frame must be processed for camera artifacts. The frame at that point is evaluated for motion and depending on the result is either discarded or accepted as having enough motion.

2.1 Implementation

The implementation was written in Java and plugged into the Java Media Framework (JMF) API through easy codec registration routines as explained in Java(TM) Media Framework API Guide [10].

The engine upon receiving the frame follows this sequence of steps:

- Calculate the reference intensity of input-image.
- Normalize the intensity values of the input-image.
- Mark intensity values (pixels) that changed based on the reference image.
- Count the blob count - the number of pixels that are composite conjoined clusters (3×3 matrix) of marked pixels.

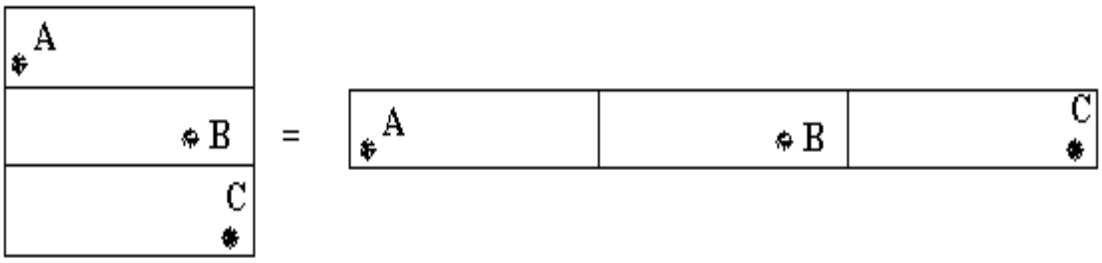

Figure 2.1: Vector representation of images; The image on the left is the original image with points A,B,C. The image on the right represents the distribution of those points in a vector.

- Apply the threshold function to the blob count and determine if the image has "enough" motion.
- Update the frame intensity and reference image.

There are some assumptions made regarding the implementation details and its syntax:

- The codec process the input frame in RGB format - Each intensity value is represented by a 3-tuple of bytes, first byte representing red color value, second - green color value, and the last - blue color value. Therefore a pure green intensity value is [0, 255, 0]. For more information about RGB format, consult Foley [2]
- Each image-frame is represented as vector. Its dimensions are $1 \times N$, where $N = width * height$ of the image (see Fig: 2.1).
- Frame-image is the frame from the camera feed being currently processed.
- Reference image is the previous frame processed.

2.2 Calculating the intensity value

The intensity of the image is the cumulative intensity value of each pixel divided by the number of pixels. We assume I to be the intensity value, P

be a pixel value, and n the is the length of the vector:

$$I = \frac{1}{n} \sum_{k=1}^{n} P(i)$$

and implemented in apollo.ares.MotionDetectionEffect.java;276-281:

```
for (ip  = 0; ip < width * height; ip++) {
    avg += (int) (inData[ip] & 0xFF);
}
avg_img_intensity = avg / outputDataLength;
```

Note that "inData" is the array containing the pixel values. It's worth noting that Java lacks the unsigned byte. Therefore any casting from byte type into any other type requires masking the signed bit.

2.3 Normalizing the input frame

The next step is to normalize the intensity of the frame-image. This is done to remove artifacts that the electronics in the camera might introduce. The algorithm has two stages.

1. Calculate the color correction value, which is the difference between the reference image intensity and the frame-image intensity (or vice-versa).

2. The difference from the reference image and the new image (for each pixel) is examined against the color correction value.

2.3.1 Color correction value

The color correction value is easily obtained from the intensity value.

The I_{img} is the intensity obtained from the current processed frame, while the I_{ref} is the reference intensity.

$$correction = \begin{pmatrix} I_{img} - I_{ref} & if I_{ref} < I_{img} \\ I_{ref} - I_{img} & if I_{img} \geq I_{ref} \end{pmatrix}$$

Where each pixel consists of a three-tuple value of colors: $P(i) = [color(i*3), color(i*3+1), color(i*3+2)]$

$$color(i) = \begin{pmatrix} color_{ref}(i) - color_{img}(i) & if(color_{ref}(i) > color_{img}(i)) < correction \\ color_{img}(i) - color_{ref}(i) & if(color_{ref}(i) \leq color_{img}(i)) < correction \\ correction & if(color_{ref}(i) > color_{img}(i)) \leq correction \\ correction & if(color_{ref}(i) \leq color_{img}(i)) \leq correction \end{pmatrix}$$

The code snippets demonstrates how this was accomplished using Java (apollo.ares.MotionDetectionEffect;292-310):

```
for (int ii=0; ii< outputDataLength/pixStrideIn; ii++) {
 refDataInt = (int) refData[ip] & 0xFF;
 inDataInt = (int) inData[ip++] & 0xFF;
 r= (refDataInt > inDataInt) ? refDataInt - inDataInt:
  inDataInt - refDataInt;
 refDataInt = (int) refData[ip] & 0xFF;
 inDataInt = (int) inData[ip++] & 0xFF;
 g= (refDataInt > inDataInt) ? refDataInt - inDataInt:
  inDataInt - refDataInt;
 refDataInt = (int) refData[ip] & 0xFF;
 inDataInt = (int) inData[ip++] & 0xFF;
 b= (refDataInt > inDataInt) ? refDataInt - inDataInt:
  inDataInt - refDataInt;

 // intensity normalization
 r -= (r < correction) ? r : correction;
 g -= (g < correction) ? g : correction;
 b -= (b < correction) ? b : correction;
...
```

2.4 Marking

The next part is to use the normalized color component values determined in the previous section to determine if the pixel changed in respect to the reference image. The root mean square of the three-color components - red, green, and blue are computed to check against the threshold value.

$$Q(i) = \sum_{i=0}^{n} \sqrt{colors(i*3)^2 + colors(i*3+1)^2 + colors(i*3+2)^2}$$

If the result is greater than the threshold value then the pixel is marked as *moved* (in our case by marking the pixel with the highest intensity value).

The code snippets shows how it was accomplished (continuation of the loop):

```
        ...
  result = (byte)(java.lang.Math.sqrt((double)
   ( (r*r) + (g*g) + (b*b) ) / 3.0));
  if (result > (byte)threshold) {
              bwData[op++] = (byte)255;
              bwData[op++] = (byte)255;
              bwData[op++] = (byte)255;
  }  else {
              bwData[op++] = (byte)result;
              bwData[op++] = (byte)result;
              bwData[op++] = (byte)result;
   }
}
```

2.5 Threshold function

The next stage is the cluster-threshold function. We count the amount of clusters (3×3 area filed with moved pixels) and determine if the count is greater than the blob threshold value. If so the frame is considered to have enough motion (see Figure2.2). Otherwise the frame is discarded.

The method to determine if the 3×3 matrix has a cluster of moved pixels is to check each pixel in that area to see if they were marked as having moved (see Fig 2.3).

Code snippet (from apollo.ares.MotionDetectionEffect.java;328-343):

```
for (op = lineStrideIn + 3; op < outputDataLength - lineStrideIn-3;
 op+=3) {
       for (int i=0; i<1; i++) {
         if (((int)bwData[op+2] & 0xFF) < 255) break;
         if (((int)bwData[op+2-lineStrideIn] & 0xFF) < 255) break;
         if (((int)bwData[op+2+lineStrideIn] & 0xFF) < 255) break;
         if (((int)bwData[op+2-3] & 0xFF) < 255) break;
         if (((int)bwData[op+2+3] & 0xFF) < 255) break;
```

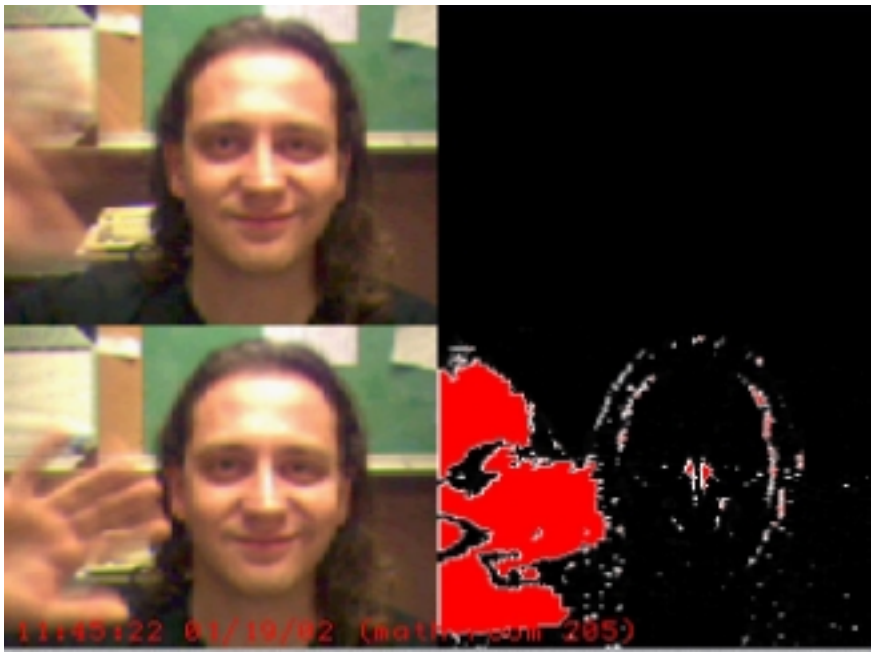

Figure 2.2: The red clusters signify the "moved" pixels

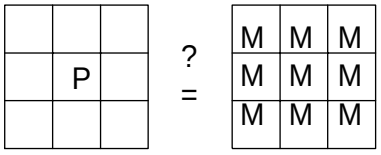

Figure 2.3: Comparison between the current tested pixel (and its surrounding pixels) against a full "motion" matrix.

```
          if (((int)bwData[op+2-lineStrideIn + 3] & 0xFF) < 255) break;
          if (((int)bwData[op+2-lineStrideIn - 3] & 0xFF) < 255) break;
          if (((int)bwData[op+2+lineStrideIn - 3] & 0xFF) < 255) break;
          if (((int)bwData[op+2+lineStrideIn + 3] & 0xFF) < 255) break;
          bwData[op]  = (byte)0;
          bwData[op+1] = (byte)0;
          blob_cnt ++;
        }
}
```

2.6 Reference intensity

The final step is to prepare for the next round of processing. Make the reference intensity be the frame intensity and make the reference image be the frame image.

2.7 Summary

The algorithm for motion detection serves to save the bandwidth usage and purge frames during inactivity periods. The implementation applies a two-stage threshold function along with intensity normalization for camera artifacts.

The algorithm is used extensively on the thin client - Ares. The algorithm was implemented in Java.

It's also worth noting that the motion engine saves substantial amount of bandwidth and storage. A typical one-day hour camera-feed (320×200 resolution, $15fps$ using medium JPEG compression) takes 151MB of space. With motion detection the space usage is about 34MB on typical busy day.

Chapter 3

Face Recognition using Eigenfaces

Face recognition technology is the integral part of Nemesis component. The face recognition technique used in Nemesis is based on Eigenfaces, described by Matthew A. Turk and Alex P. Pentland in their paper titled "Face Recognition Using Eigenfaces." [14]

The Eigenface technique belongs to template matching family. As such it computes a set of *hash values* for each image. The hash value of the image we wish to match is then compared to the hash values of images in the database encoded similarly. The authors in their paper explain that much of the previous work in face recognition has ignored the issue of what aspects of face are important for identification. Therefore encoding and decoding face images using local and global features of our face in which features, such as nose, eyes, ears may or may not be related to face recognition. A simple approach to extract the information is to capture the variation in a collection of face images, independent of any judgment of features. In mathematical terms, the authors say, the algorithm finds the principal components of distribution of faces, or the eigenvectors of the covariance matrix of the set of face images. These eigenvectors can be thought of as a set of features that together characterize the variation between face images. Each image location contributes more or less to each eigenvector as a sort of ghostly face that they call an *eigenface* (see Fig: 3.1).

Furthermore the authors mention that also the face images can be reconstructed by weighted sums of a small collection of characteristic faces. From which an efficient way to learn and recognize faces might be to build the

Figure 3.1: Eigenface

characteristic features from known face images and to recognize particular faces by comparing the feature weights needed (approximately) to reconstruct them with weights associated with the known individuals.

This is exactly what was implemented in the scalable distributed face recognition framework.

There are three steps in using this algorithm:

- Initialization of the training images, calculate "eigenfaces" and their respective weights.
- Compare the given image.
- Determine if the given image is sufficiently close to a face in the face space.

3.1 Initialization

There are eight steps in constructing face-space:

- Construct a face vector with the training images.

- Calculate the average face.
- Normalize the training images.
- Compute eigenvector and eigenvalues.
- Extract the MAGICNR (which is some number, in the implementation it's eleven) most significant eigenvalues (and their respective eigenvectors).
- Project the eigenvectors onto the face vector, result being the face-space.
- Normalize the face vector.
- Calculate the set of weights associated with each training image.

There are certain assumptions made in the algorithm:

- Every image's dimensions are the same.
- Each image is represented as a vector component. The vector dimensions are $1 \times N$, where we assume $N = width * height$ (see Fig: 2.1).

3.1.1 Training images

The paramount step is constructing a face vector that consist of sixteen training models (images). A matrix of dimension $16 \times N$ is constructed (N is the $width * height$ of the image), and each row is an image and each column is an intensity value of the image (see Fig: 3.2).

And the code snippet (from apollo.nemesis.FaceFaceCreator.java:277-280) demonstrates how the image (which nativly is represented in a vector format) is copied to the $16 \times N$ array.

```
double[][] face_v = new double[16][width*height];

for (i = 0; i < files.length; i++) {
       face_v[i] = files[i].getDouble();
}
```

Where "files" is a JPGFile (or PPMFile) object and "getDouble()" returns the representation of the image in RGB format as a vector.

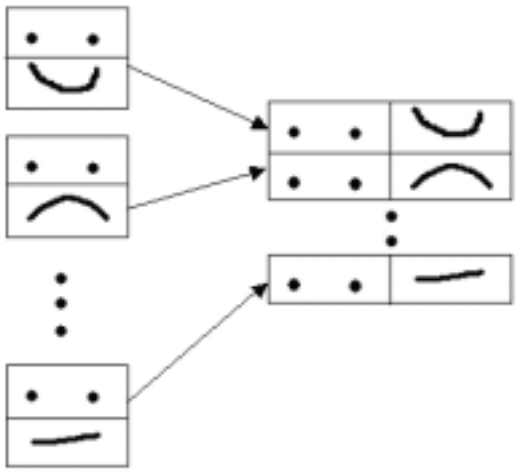

Figure 3.2: How images are constructed in a face vector

3.1.2 Average Face

The average face is computed, which is a vector where each column determines the average intensity across the sixteen (M) images.

Where M is sixteen, I is one of the M's images and A is the average face vector.

$$A(j) = \frac{1}{M} \sum_{i=0}^{M} I_i(j)$$

Demonstrated by the code (from apollo.nemesis.EigenFaceComputation.java: 85-96):

```
double[] avgF = new double[length];
for ( pix = 0; pix < length; pix++) {
  temp = 0;
  for ( image = 0; image < nrfaces; image++) {
    temp +=  face_v[image][pix];
  }
  avgF[pix] = temp / nrfaces;
}
```

Where "length" is the N (the $width * height$ of the image). "nrfaces" is N.

3.1.3 Normalization

The average face is subtracted from the face-vector ($M \times N$ dimension, with M images) to normalize the images using:

$$I_i(j) = I_i(j) - A(j)$$

where:

$0 \leq j < N; 0 \leq i < M$

Demonstrated by this code snippet (from apollo.nemesis.EigenFaceComputation.java:104-109):

```
for ( image = 0; image < nrfaces; image++) {
  for ( pix = 0; pix < length; pix++) {
    face_v[image][pix] = face_v[image][pix] - avgF[pix];
  }
}
```

3.1.4 Eigenvalues and eigenvectors

The next step is to compute the hash values, so called eigenvector and eigenvalues. The eigenvalues and eigenvectors are characteristics values of a matrix. From the Castleman "Digital Image Processing" [1]:

> Each eigenvalue can be thought of as an amount which, when subtracted from each diagonal element, makes the matrix singular... Eigenvectors are characteristic vectors of the matrix. Each [eigenvector] corresponds to one of the eigenvalues.

Therefore, we are looking for the v (eigenvectors) and λ (eigenvalues) defined as:

$$Av = \lambda v$$

For example, suppose:

$$A = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$$

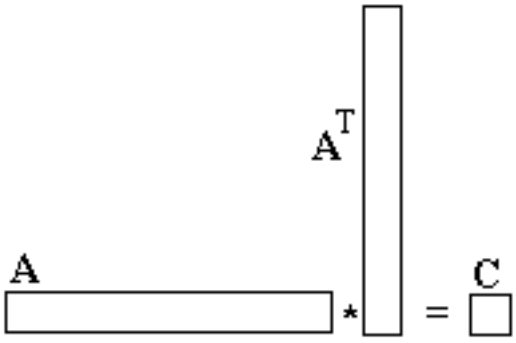

Figure 3.3: Covariance matrix

$$A\begin{pmatrix} 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 3 \end{pmatrix} = 3\begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Therefore, 3 is the eigenvalue of this A matrix with $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ being the eigenvector.

Computing these values is an extremely intensive task for typical images sizes. Fortunately we can determine the eigenvalues and eigenvectors by solving a much smaller $M \times M$ matrix problem and the take the linear combination of the result. More details of why we can do this is explained in Castleman [1], et al. We use the covariance matrix to compute the eigenvalues and eigenvectors using Jacobian transformation or Singular Value Decomposition depending on the properties of the covariance matrix [3, 5, 7, 6]. Discussing the details of these techniques is beyond the scope of this thesis. The implementation of eigenvalue and eigenvector decomposition is detailed explained in "Numerical Recipies in C: The art of scientific computing" by Press et. al. [5].

$$C = A^T A$$

or see Figure 3.3:

Our eigenvalues after the calculations are in random distribution. To find the MAGICNR most extreme values we need to sort the eigenvalues and its

corresponding eigenvectors. This is done using a modified quick sort method and repositioning the eigenvectors in their appropriate columns (depending on the sorted eigenvalues).

As illustrated by this code snippet (from apollo.nemesis.EigenFaceComputation.java:145-171):

```
    int[] index = new int[nrfaces];
    double[][] tempVector = new double[nrfaces][nrfaces];
/* Temporary new eigVector */

    for ( i = 0; i <nrfaces; i++) /* Enumerate all the entries */
      index[i] = i;

    doubleQuickSort(eigValue, index,0,nrfaces-1);
    // Put the index in inverse
    int[] tempV = new int[nrfaces];
    for ( j = 0; j < nrfaces; j++)
      tempV[nrfaces-1-j] = index[j];
    index = tempV;
     /*
      * Put the sorted eigenvalues in the appropiate columns.
     */
    for ( col = nrfaces-1; col >= 0; col --) {
      for ( rows = 0; rows < nrfaces; rows++ ){
        tempVector[rows][col] = eigVector[rows][index[col]];
      }
    }
    eigVector = tempVector;
```

3.1.5 Projection on to face space

Multiplying the sorted eigenvector with our face vector results in getting the face-space vector. This gives us the same result as if we had performed the eigenface computations on the face vector itself and not on the covariance matrix. Details of why we can do this is explained in [13].

3.1.6 Normalize

Normalizing the face-space is a simple procedure. The maximum of the face-space is divided by each pixel of the face-space.

$$Q(i) = \frac{P(i)}{max(P)}$$

and demonstrated by the code snippet (from apollo.nemesis.EigenFaceComputation.java: 182-193):

```
Matrix eigVectorM = new Matrix(eigVector, nrfaces,nrfaces);
double[][] faceSpace = eigVectorM.times(faceM).getArray();
eigVector = null;
for ( image = 0; image < nrfaces; image++) {
 temp = max(faceSpace[image]); // Our max
 for ( pix = 0; pix < faceSpace[0].length; pix++)
   faceSpace[image][pix] = Math.abs( faceSpace[image][pix] / temp);
}
```

3.1.7 Weights

The final step is to calculate the set of weights associated with the face space. Each weight is a vector off dimension $1 \times MAGICNR$. The weights are the result of multiplying the transpose of each row from face-space vector with the normalized training images.

The implementation calculates M weights: each training image has an associated weight vector of length $MAGICNR$ with each other image. Therefore each training image weight vector represents a $MAGICNR$-dimension (in the implementation its 11t-dimension) vector. Each value in the weight vector represents the "similarity" to the other M training images.

This simple computation is done (from apollo.nemesis.EigenFaceComputation.java: 204-217):

```
double[][] wk = new double[nrfaces][MAGIC_NR]; // M rows, 11 columns
for (image = 0; image < nrfaces; image++) {
   for (j  = 0; j <  MAGIC_NR; j++) {
     temp = 0.0;
     for ( pix=0; pix< length; pix++)
```

```
        temp += faceSpace[j][pix] * faces[image][pix];
      wk[image][j] = Math.abs( temp );
    }
  }
```

This completes the final stage of initializing the face recognition models.

3.1.8 Summary

The calculations to obtain the weights, face-space are computationally intensive. Therefore this task is only done once for the set of images. In the implementation the results are cached so the computation steps can be skipped the next time Nemesis is started.

3.2 Recognition

Recognition of an image is rather a simple task compared to the first stage.

There are three steps in recognition:

- Transform the input image into eigenface components (project it onto the face-space).
- Calculate the input image weights.
- Determine the Euclidian distance of the input image weights to the weights of the set of images from the face-space.
- Determine (based on the Euclidian distance and on the threshold value) if the input image is matched against the database of images.

3.2.1 Transform

Projecting the input image onto the face space means:

1. Subtract the average face from the input face (to normalize the image).
2. Project the normalized image onto the face space.

Normalizing the input face is a simple technique. We use the given average face vector (computed earlier) to subtract each intensity value from the given image:

$$I(j) = I(j) - A(j)$$

where:

$0 \leq j < N$

Project the normalized image onto the face space consist of multiplying the given image from the face-space with the normalized input image. Fortunately this is also the step where we determine the weight of the image.

This computation is illustrated by the code snippet (from apollo.nemesis.FaceBundle.java: 213-223):

```
double[] input_wk = new double[MAGIC_NR];
double temp = 0;
for (j = 0; j < MAGIC_NR; j++) {
  temp = 0.0;
  for ( pix=0; pix <length; pix++)
    temp += faceSpace[j][pix] * inputFace[pix];

  input_wk[j] = Math.abs( temp );
}
```

Where the "inputFace" is the vector representing the image.

3.2.2 Euclidian distance

Euclidian distance is the cumulative difference of each index in a vector. In our case we calculate the distance on the input image weights and our training-image-weights.

The code snippet belows illustrates the computation (from apollo.nemesis.FaceBundle.java: 229-246):

```
double[] distance = new double[MAGIC_NR];
double[] minDistance = new double[MAGIC_NR];
idx = 0;
for (image = 0; image < nrfaces; image++) {
    for (j = 0; j < MAGIC_NR; j++) {
```

```
          distance[j] = Math.abs(input_wk[j] - wk[image][j]);
        }
  ....
```

3.2.3 Thresholding

The final step is to determine if the normalized distance is less than the threshold value. The image is considered recognized if the distance value is less then the threshold value.

```
        .....
        if (image == 0)
          System.arraycopy(distance,0,minDistance,0,MAGIC_NR);
        if (sum(minDistance) > sum(distance)) {
          this.idx = image;
          System.arraycopy(distance,0,minDistance,0,MAGIC_NR);
        }
    }
    if (max(minDistance) > 0.0)
      divide(minDistance, max(minDistance));

    minD = sum(minDistance);
```

Based on the minD and the global threshold value its determined if the input image is matched against the set of training images.

3.3 Summary

The face recognition technique used in this work is based on Eigenface technique described by Turk and Pentland in their work. It was implemented in Java for portability purpose and easy of reading. The recognition technique is paramount in the Nemesis component which carries out the face-recognition task on the camera feed provided by Ares.

Chapter 4

Dynamic aspects

The agility and adaptability is a requirement for a scalar distributed system. The system must be flexible and capable of addressing various problems: network loss connectivity, power outage, demand for more components, notification of new services and switching new load onto them, and many more. All of this must be handled in a reliable distributed system. These are a must for a true distributed system.

4.1 Interactions of a distributed system

The most essential interactions in this distributed system are:

- Notification and registration of new components.
- Querying the components for its load and availability.
- Finding components with the least load.
- Discontinuing the use of deceased components and using new ones.
- Work dynamically.

4.2 Notification and registration of new components

Each component, except Aries, whenever they are started notifies the other components about its presence. The only component that takes notice of it

this is Hermes. Hermes, being the load balancer needs to know the whole set of network components.

The method by which the components find out about each other presence is by using Jini technology - mainly leveraging the multicast request protocol as described in Jini(TM) Architecture Specifications [12]. The method by which a component announces its presence is by locating the lookup services (which are native to Jini), download code to control the lookup service, use that code to register itself (and also upload its own code) and then periodically renew the registration. The code that is uploaded includes simple information that can be modified and queried - mainly the count of users, the maximum amount of users that can be handled, and the address of the system. This information is used in finding the load and availability of the system, which is explained in the next section.

All of these components: Hermes, Mors, Nemesis and Demeter are services (in Jini terminology), while Ares is the client. All the services are using Jini to announce their presence and find, if needed, the other components (Nemesis looks for Mors using Hermes' knowledge). Ares on the other hand, being a client doesn't announce its presence - it searches for the services it requires.

4.3 Querying the components for its load and availability

After the service components have been registered with the lookup server, Hermes queries each new found components for its information (see Fig 4.1). It does that every predefined amount of time. This allows for retrieval of near-real time statistical information on the load of each service. It also allows for discovering if the service has been disconnected or is no longer operational and accordingly purge information about the service.

Only Hermes queries for these information. All other components just provide the pertinent information and change their information accordingly to their status.

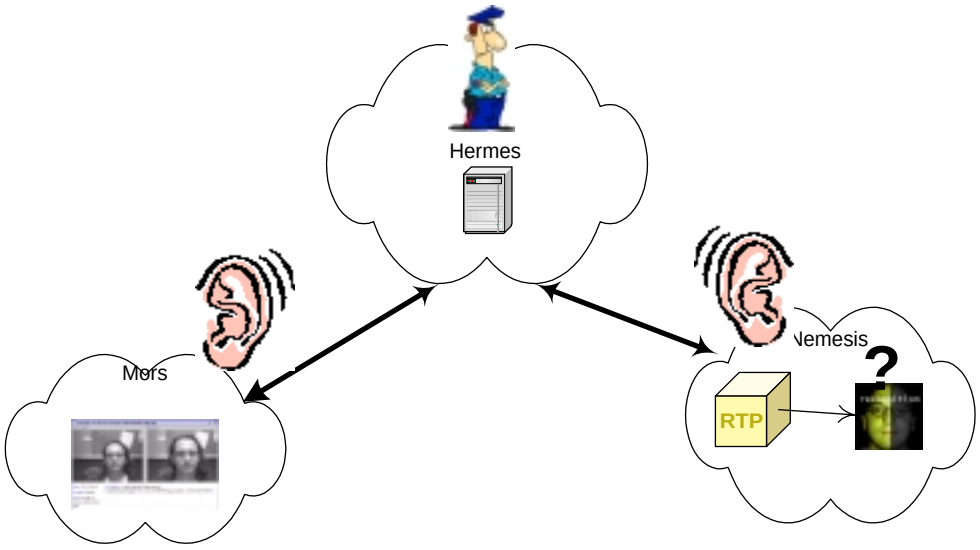

Figure 4.1: Information querying by Hermes

4.4 Finding components with the least load

Ares and Nemesis are two of the components that require access to the other components. Ares requires Demeter and Nemesis, while Nemesis requires Mors. Each of these clients needs to find the appropriate service.

The requesting client queries Hermes, which knows the least populated service in the desired pool. Hermes provides the address to the least loaded service and the requesting client uses that address to talk to the service directly. If the requested component is not available, no address is returned.

It is assumed to that the components can and will stop working at some point. Therefore the connection between the components can break at any time and should be re-established. If there are no desired components at the current time then the service should continue asking Hermes for that component repeatedly until its found.

When the required component is found it's address is cached and periodically checked. This makes it possible to discover dead services and request new ones from Hermes. Vice-versa - if the connection is ok, there is no need query Hermes for a least loaded service in the pool

4.5 Work dynamically

With the idea of notification, registration, checking the components its feasible to adjust to changing network conditions. New services can be taken advantage of and other nodes in a pool can be shutdown for maintenance. All these features allows for flexible rollover off services. In turn making the whole system capable of working truly dynamic scalable distributed fashion.

4.6 Summary

All of the requirements demanded by a scalable distributed framework have been implemented in the work. The underlying technology used to discover, notify, and register components used was Jini. Remote Method Invocation (RMI [8]) was used to check the load and availability of each of the services, along for exchanging specific information with components.

Bibliography

[1] N.M. Allinson, A.W. Ellis, B.M. Flude, and A.J Luckman. A connectionist model of familiar face recognition. In *IEE Colloquium on Machine Storage and Recognition of Faces*, pages 1–10, 1992. Digest No: 1992/017.

[2] R. Brunelli and T. Poggio. Caricatural effects in automated face perception.

[3] R. Brunelli and T. Poggio. Face recognition through geometrical features.

[4] R. Brunelli and T. Poggio. Hyberbf networks for gender classification.

[5] R. Brunelli and T. Poggio. Hyperbf networks for real object recognition. In *Proc. of the 12th IJCAI*, pages 1278–1284, Sidney, Australia, 1991.

[6] Kenneth R. Castleman. *Digital Image Processing*. Prentice-Hall, Inc., 1996.

[7] Ross Cutler. Face recognition using infrared images and eigenfaces. April 1996.

[8] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics Principle and Practice*. Addison-Wesley, Inc., 1997.

[9] Francis Galton. *Personal Identification And Description*. June 1888.

[10] Gaston Gonnet. Singular value decomposition and eigenvalue decomposition. November 2001.

[11] Audio-Video Transport Working Group, H. Schulzrinne, GMD Fokus, S. Casner, Precept Software Inc., R. Frederick, Xerox Palo Alto Research Center, V. Jacobson, and Lawrence Berkeley National Laboratory. Rtp: A transport protocol for real-time applications. January 1996.

[12] William H., Teukolsky Saul A., Vetterling William T., and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1992.

[13] L.D. Harmon, M.K. Khan, R. Lasch, and P.F. Ramig. Machine identification of human faces. *Pattern Recognition*, 1981.

[14] Jim Hefferon. *Linear Algebra.*

[15] T. Kanade. Computer recognition of human faces. *Interdisciplinary Systems Research*, 1977. Birkhauser Verlag.

[16] Mohamed Amine Khamsi. Eigenvalues and eigenvectors technique.

[17] T. Kohonen. *Self-organization and associative memory.* Springer-Verlag, 1988. 2nd Edition.

[18] Sun Microsystem. *Java(TM) Remote Method Invocation Specification.* 1998.

[19] Sun Microsystems. *JMF Frequently Asked Questions.*

[20] Sun Microsystems. *Java(TM) Media Framework API Guide.* November 1999.

[21] Sun Microsystems. *Jini Network Technology Datasheet.* May 2001.

[22] Sun microsystems. *Jini(TM) Architecture Specification.* Sun Microsystems, 2001.

[23] L. Najman, R. Vaillan, and E. Pernot. Face from sideview to identification. In G. Vernazza, A.N. Venetsanopouls, and C. Braccini, editors, *Image Processing: Theory and Applications.* Elsevier Science Publishers, 1993.

[24] O. Nakamura, S. Mathur, and T. Minami. Identification of human faces based on isodensity maps. *Pattern Recognition*, pages 263–272, 1991.

[25] Alexander Pentland and Terrence Sejnowski. Neural networks and eigenfaces for finding an analyzing faces.

[26] A. Samal and P.A. Lyengar. Automatic recognition and analysis of human faces and facial expressions: A survey. *Pattern Recognition*, 1992.

[27] M. Turk and A. Pentland. Eigenfaces for recognition. In *Journal of Cognitive Neuroscience*, March 1991.

[28] Matthew A. Turk and Alex P. Pentland. Face recognition using eigenfaces. May 1991.

[29] K.H. Wong, H.H.M. Law, and P.W.M. Tsang. A system for recognising human faces. In *Proceedings of the Internation Conference on Acoustics, Speech and Signal Processing*, pages 1638–1642, 1989.

[30] C.J. Wu and J.S. Huang. Human face profile recognition by computer. *Pattern Recognition*, pages 255–259, 1990.